AFRL-IF-RS-TR-2003-259
Final Technical Report
November 2003

A JAVA-BASED ACTIVE NETWORK OPERATING SYSTEM (JANOS)

University of Utah

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. H136

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 074-0188*

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** NOVEMBER 2003	**3. REPORT TYPE AND DATES COVERED** Final Jun 99 – Feb 03

4. TITLE AND SUBTITLE A JAVA-BASED ACTIVE NETWORK OPERATING SYSTEM (JANOS)	**5. FUNDING NUMBERS** C - F30602-99-1-0503 PE - 62301E PR - H136 TA - 00 WU - 01
6. AUTHOR(S) Jay Lepreau	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Utah School of Computing 50 South Central Campus Drive Salt Lake City Utah 84112-9205	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington Virginia 22203-1714 AFRL/IFGA 525 Brooks Road Rome New York 13441-4505	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2003-259

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Paul Sierak/IFGA/(315) 330-7346/ Paul.Sierak@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 Words)***

The JANOS projects' primary objective was to develop a principle local operating system for active network nodes. The operating system is oriented toward executing untrusted Java bytecode, primarily for management and control.

14. SUBJECT TERMS Active Networks, JAVA Operating System	**15. NUMBER OF PAGES** 17
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

Contents

1 Overview

The objective of the Janos project was to develop a principled local operating system for active network nodes, oriented toward executing untrusted Java bytecode. The primary focus was resource management and control, with a secondary objective of performance. Janos also provided technology transfer of broadly and separately useful software components, in Open Source form.

Approach: Janos is a specialized but full-fledged operating system. Its primary target applications are untrusted Java bytecode programs, but different levels of Janos can also execute several other types of trusted and untrusted programs, written to different APIs, and in different languages. Janos is structured as a single-address space operating systems, implemented primarily in the C and Java languages, containing a *custom Java virtual machine*, the *JanosVM*. Instead of hardware-based memory protection, primary memory protection derives from the safety properties of the Java language. However, unlike standard Java-based systems, robust protection and control is provided by the JanosVM which supports inter-application isolation within its sole instance.

Janos leverages experience providing *resource control* in more traditional operating systems, including extending the notion of a user/kernel boundary to a typesafe language environment. Janos supports controls from the lowest levels, through the virtual machine, to applications. This includes obvious resources such as CPU cycles, network bandwidth, and memory, and may include less obvious ones such as caching store, persistent disk store, specialized hardware, and specialized data such as routing table entries. Resource control, resource utilization, and accurate admission control are enhanced by the ability of Janos to load both trusted native code components and verifiable components written in Cyclone, a safe dialect of C. These components are assembled into flexible ”Click”-like router graphs for fast packet processing.

Underneath the JanosVM is *Moab*, a *custom active network NodeOS*. Moab builds on the Utah *OSKit*, a large set of reusable components for constructing OS-like entities. The OSKit was extended with additional components needed for a secure active network node such as controls on processor and network use. Moab and the OSKit allow Janos to run directly on bare hardware or atop standard Unix—thereby enhancing technology transfer.

Janos and its components demonstrated their worth in real application domains such as censor-resistant publishing and distributed multiplayer simulation. Partially supported by this grant, we also developed a large-scale *network testbed* called Emulab with configurable control of the nodes, links, and topology. This testbed, also available to researchers worldwide, provided a unique research instrument on which to evaluate Janos, other NodeOS implementations, and complex active protocols.

Although the primary target of Janos is active networks, the project was structured to have *broad relevance and impact*. It produced separately useful NodeOS, Java VM, and active network runtime components. The JanosVM, when customized with a domain-specific runtime, can provide strong resource control and security wherever Java runs. As another route to broad impact, active involvement with *specification efforts* in both the active network and Java realms were goals of this project, successfully accomplished.

Software Releases and Service: During the course of this project, we made a large number of formal software releases, for all the components. For example, our records indicate that during 2000–2002 we made at least 29 formal, documented software releases. From October 2000 on, we also made our testbed publically available and had hundreds of external researchers using it by the end of the Janos grant. For example, even by July 2001 seven other projects in the DARPA Active Networks program were using it.

Scope, Funding, and Schedule: This project was originally budgeted at $2,826,411, but due to several additional tasks (two technology integration demonstrations and responsibility for the maintainance/-enhancement/replacement of the "ANTS" execution environment), received an additional $334,784, for a total of $3,161,195. The additional tasking caused the grant duration to be extended another eight months.

Outline: The bulk of the rest of this report is structured around the technical areas, software systems, or specifications explored or developed under this project. They fall into seven categories. The report concludes with a list of publications.

1. The Janos Active Network Operating System
2. Java Operating Systems
3. Other Technology Components of Janos
4. Active Network Execution Environments
5. Active Applications
6. Language and Compiler Technology
7. Emulab/Netbed: an Integrated Network Testbed

2 The Janos Active Network Operating System

First we give a quick overview of the Janos architecture and its layers. Janos is an operating system for active network nodes whose primary focus is strong resource management and control of untrusted active applications written in Java. In a paper [1] that covered the whole system, we laid out the Janos design and its rationale.

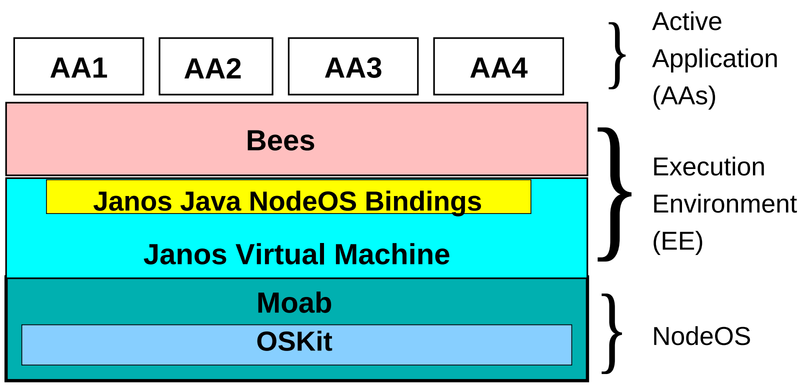

Figure 1: Janos Layers

The Janos Layers

Janos includes the three major components of a Java-based active network operating system: the low-level NodeOS, a resource-aware Java Virtual Machine, and an active network protocol execution environment. In reality, Janos is composed of five distinct software components, all of which are separately usable.

- *The OSKit:* A framework and a set of 34 component libraries oriented to operating systems, together with extensive documentation.
- *Moab:* The layer implementing the "NodeOS" operating system API, built on the OSKit.
- *JanosVM:* A resource-aware, multi-process Java Virtual Machine. It can be tailored to and take advantage of the Moab abstractions, resource and security interfaces.
- *Java NodeOS:* An implementation of the NodeOS abstractions and services in Java. It refines the interface for Java, eliminating non-Java abstractions (such as address spaces), and adding support for Java-isms (such as Classes).
- *Execution Environments:* Finally, Janos supports two distinct Execution Environments, the Active Network Transport System (ANTS) and the more sophisticated Bees environment.
  - *ANTS v2.0:* An enhanced version of the existing ANTS architecture that provides primitive isolation and resource controls for active applications.
  - *Bees:* A newly developed system that provides a rich environment for mobile code, a flexible authentication and authorization mechanism, capability-based access to privileged resources, and services for discovering and monitoring neighboring nodes.

3 Java Operating Systems

A fundamental part of our work was exploring the implications of applying OS structuring concepts to operating systems that use the type-safe properties of the Java programming language to provide memory safety, instead of using the hardware MMU to do so. We explored many aspects of the issues in numerous experimental prototypes, with an emphasis on *resource control*.

3.1 Making the Case: OS Structure for Mobile Code

In our first effort in this domain, we designed and implemented "Alta," an implementation of the highly-structured Fluke [2] nested process model, but in a Java virtual machine instead of supported by hardware-based memory protection.

In a highly competitive and visible OS workshop, we presented and published early results from our Alta prototype [3], focusing on making the case for such a comprehensive approach. The majority of work on protection in single-language mobile code environments focuses on information security issues and depends on the language environment for solutions to the problems of resource management and process isolation. We argued that what is needed in these environments are not ad-hoc or incremental changes but a coherent approach to security, failure isolation, and resource management. Protection, separation, and control of the resources used by mutually untrusting components, applets, applications, or agents are exactly the same problems faced by multi-user operating systems. We argued that real solutions will come only if an OS model is uniformly applied to these environments. We presented *Alta*, our prototype Java-based system patterned on Fluke, a highly structured, hardware-based OS, and reported on its features appropriate to controlling mobile code. Several ideas from this work were then folded into the JanosVM, most importantly, a method for sharing classes between processes.

3.2 Modeling an Existing OS: The Alta Operating System

Many modern systems, including web servers, database engines, and operating system kernels, use language-based protection mechanisms to provide the safety and integrity traditionally supplied by hardware. As these language-based systems become used in more demanding situations, they are faced with the same problems that traditional operating systems have solved—namely shared resource management, process separation, and per-process resource accounting. While many incremental changes to language-based, extensible systems have been proposed, we demonstrated in an implementation and a thesis [4] that comprehensive solutions used in traditional operating systems are applicable and appropriate.

The thesis gives a detailed description of Alta, an implementation of the Fluke operating system's nested process model in a Java virtual machine. The nested process model is a hierarchical operating system process model designed to provide a consistent approach to user-level, per-process resource accounting and control. This model accounts for CPU usage, memory, and other resources through a combination of system primitives and a flexible, capability-based mechanism.

Alta supports nested processes and interprocess communication. Java applications running on Alta can create child processes and regulate the resources—the environment—of those processes. Alta demonstrates that the Java environment is sufficient for hosting traditional operating system abstractions. Alta extends the nested process model to encompass Java-specific resources such as class files, modifies the model to leverage Java's type safety, and extends the Java type system to support safe fine-grained sharing between different applications. Existing Java applications work without modification on Alta.

Alta was compared in terms of structure, implementation and performance to Fluke and traditional hardware-based operating systems. A small set of test applications demonstrated flexible, application-level control over memory usage and file access.

3.3 Comparing Techniques: Surveying the Design of Java Operating Systems

Language-based extensible systems, such as Java Virtual Machines and SPIN, use type safety to provide memory safety in a single address space. By using software to provide safety, they can support more efficient IPC. Memory safety alone, however, is not sufficient to protect different applications from each other. Such systems need to support a *process model* that enables the control and management of computational resources. In particular, language-based extensible systems should support resource control mechanisms analogous to those in standard operating systems. They need to support the separation of processes and limit their use of resources, but still support safe and efficient IPC.

In a paper comparing very different systems we demonstrated how this challenge is being addressed in three Java-based systems [5]. First, we laid out the design choices when implementing a process model in Java. Second, we compared the solutions that have been explored in several projects: Alta, K0, and the J-Kernel. Alta closely models the Fluke operating system; K0 is similar to a traditional monolithic kernel; and the J-Kernel resembles a microkernel-based system. We compared how these systems support resource control, and explore the tradeoffs between the various designs.

3.4 Isolating a Key Issue: Drawing the Red Line in Java

Software-based protection has become a viable alternative to hardware-based protection in systems based on languages such as Java, but the absence of hardware mechanisms for protection has been coupled with an absence of a user/kernel boundary. In a well-received paper [6] we showed why such a “red line” must be present in order for a Java virtual machine to be as effective and as reliable as an operating system. We discussed how the red line can be implemented using software mechanisms, and explained the ones we use in the Java system that we are building.

The red line was originally implemented in KaffeOS in a fashion similar to traditional operating systems. User processes requesting system services would cross the line into a kernel process where the request was guaranteed to complete, regardless of asynchronous termination requests. Boundary crossings and system services were then implemented directly in Java and colocated with user processes. Colocation of privileged system code and objects with untrusted user code leads to benefits like increased performance and improved accuracy in accounting of resources. Limiting access to the privileged parts of the system was done by leveraging Java’s type-safety and fine-grained object access permissions.

3.5 Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java

Single-language runtime systems, in the form of Java virtual machines, are widely deployed platforms for executing untrusted mobile code. These runtimes provide some of the features that operating systems provide: inter-application memory protection and basic system services. They do not, however, provide the ability to isolate applications from each other, or limit their resource consumption. We designed and implemented KaffeOS, a Java runtime system that provides these features, and described its key aspects and their rationale in great detail in a paper [7] and a thesis [8]. The KaffeOS architecture takes many lessons from operating system design, such as the use of a user/kernel boundary, and employs garbage collection techniques, such as write barriers.

The KaffeOS architecture supports the OS abstraction of a process in a Java virtual machine. Each process executes as if it were run in its own virtual machine, including separate garbage collection of its own heap. The difficulty in designing KaffeOS lay in balancing the goals of isolation and resource management against the goal of allowing direct sharing of objects. Overall, KaffeOS is no more than 11% slower than the freely available JVM on which it is based, which is an acceptable penalty for the safety that it provides. Because of its implementation base, KaffeOS is substantially slower than commercial JVMs for trusted code, but it clearly outperforms those JVMs in the presence of denial-of-service attacks or misbehaving code.

3.6 A Flexible Base: the JanosVM Java Virtual Machine

The JanosVM is a Virtual Machine that can execute Java byte codes in multiple isolated contexts. The goal of JanosVM is not to implement a full Java Operating System, rather, it aims to provide the components used to implement a Java OS. For example, there is basic functionality for interacting with concurrently running processes, but no IPC system is directly exported to user code. Instead, an arbiter library can be developed which maps the JanosVM primitives to a safer API, for example, a message passing system. It is hoped that this approach affords more freedom for experimentation and specialization without requiring each developer to roll their own JVM.

Our previous experiences with Java OSs served as the basis for much of the JanosVM's design. KaffeOS's code base and overall design became the starting point, while thread migration and Alta's flexible typing provided some interesting variation. However, there were several changes in the final implementation of the JanosVM, usually in favor of making the system more resilient in the face of faults. First, the KaffeOS garbage collector was changed to drop a processes' memory instead of merging it into the kernel's heap. This removed the possibility of process memory allocated by native code leaking into the kernel, an occasional source of grief while working on KaffeOS. Next, thread migration was used to replace the transition between user processes and the kernel, thus allowing threads to switch to peer or server processes. Because the system lies in a single hardware address space, there is no significant additional overhead in this change and it enables the use of a microkernel-like design. Finally, a modified version of the Alta type system was added to enable simple class sharing as well as type renaming. This addition allowed JanosVM to continue to use a modified version of the java.lang package without precluding the possibility of a true JDK compatible version.

3.7 Making it Standard: JSR-121 Java Isolates

Based on its active network research, the University of Utah was selected as an official representative on the Expert Group associated with the Sun "Java Specification Request" for functionality that is slated to become part of standard Java in the future. This is JSR-121, the "Application Isolation API Specification," for initiating and controlling computations isolated from each other. The scope of JSR-121 is limited to isolation. However, this JSR is the beginning of a process in which much of the Java application isolation and control that we researched and developed will eventually find its way into standard Java. JSR-121 is crucially important: it clear isolation semantics, which Java previously lacked, will directly lead to more robust and secure Java-based systems,

Not only did we take a leading role in the multi-year specification effort, but we demonstrated JanosVM's power and flexibility by quickly providing a prototype implementation of the initial JSR-121 "straw man" API. Furthermore, we provided the first publicly available implementation of the API submitted for public review.

4 Other Technology Components of Janos

4.1 An OS Interface for Active Routers

This paper [9] described an operating system interface for active routers. This interface allows code loaded into active routers to access the router's memory, communication, and computational resources on behalf of different packet flows. In addition to motivating and describing the interface, the paper also reports our experiences implementing the interface in three different OS environments: Scout, the OSKit, and the exokernel.

We contributed a great deal to the development of a common set of abstractions for the lowest layer of an active router, the NodeOS, and helped document them in specification document [10]. This interface allows code loaded into active routers to access the router's memory, communication, and computational resources on behalf of different packet flows. We refined and validated the Active Network community's NodeOS specification through the on-going implementation of Moab, our implementation of the NodeOS API on the OSKit. We drove the discussion on changes required for the channel and demultiplex key specifications, memory models, threading, and file systems. Through our implementation, we validated the threading model and the basic channel design.

Moab is built upon the OSKit, from which Moab gets a full complement of device drivers (originally from Linux), several complete filesystems, a threading implementation, the FreeBSD networking stack, and many support modules such as boot loaders and remote debugging support. One benefit of building on the OSKit is that POSIX-reliant systems will work almost immediately on the OSKit, and can be incrementally migrated to Moab. The NodeOS API is implemented as a library layered on top of the OSKit's libraries. Moab is not implemented as a traditional operating system; invocations of NodeOS functions are direct function calls and do not "trap" into the OS. (Janos protects itself from untrusted code at the JanosVM layer, not in the NodeOS.) Moab, like the OSKit, is written in C.

Moab can also be layered on top of a POSIX system in place of the OSKit, obviating the need for a dedicated machine on which to run and simplifying development and debugging. However, in such a configuration, some functionality is lost, most notably precise resource control. Currently configurations exist for FreeBSD, Linux and Solaris.

4.2 Hybrid Resource Control of Active Extensions

Using Java provides a great deal of flexibility in Janos; however, it is not always sufficient to satisfy the raw forwarding performance needs of some applications. In support of this, we explored alternative means of extensibility that could be used in conjunction with the Java-based components. Our approach, outlined in a paper [11], was to include an existing technology, MIT's Click modular router, in the lowest layers of the system and then address the safety concerns that arise, both memory safety and resource safety.

The ability of active network technology to allow customized router computation critically depends on having resource control techniques that prevent buggy, malicious, or greedy code from affecting the integrity or availability of node resources. It is hard to choose between static and dynamic checking for resource control. Dynamic checking has the advantage of basing its decisions on precise real-time information about what the extension is doing but causes runtime overhead and asynchronous termination. Static checking, on the other hand, has the advantage of avoiding asynchronous termination and runtime overhead, but is overly conservative. This work presents a hybrid solution: static checking is used to reject extremely resource-greedy code from the kernel fast path, while dynamic checking is used to enforce overall resource control.

This hybrid solution reduces runtime overhead and avoids the problem of asynchronous termination by delaying extension termination until times when no extension code is running, i.e., between processing of packets.

We designed and created an initial implementation of the key parts of a hybrid resource control technique, called RBClick. RBClick is an extension of the Click modular router, customized for active networking in Janos, an active network operating system. RBClick uses a modified version of Cyclone, a type-safe version of C, to allow users to download new router extensions directly into the Janos kernel. Our measurements of forwarding rates indicate that hybrid resource control can improve the performance of router extensions by up to a factor of two.

4.3 The Java NodeOS

This is an implementation of the NodeOS abstractions and services, implemented in Java. It refines the interface for Java, eliminating non-Java abstractions (such as address spaces), and adding support for Java-isms (such as Classes). The Java NodeOS allows a Java-based EE author to build a NodeOS-compliant EE, and run it on a regular JVM *or* on the JanosVM. ANTS2 is an example of an EE written to the Java NodeOS API.

Our Java NodeOS implementation is portable to any Java system. It was used by many other institutions in the Active Networks program, and numerous others.

4.4 Knit: Component Composition for Systems Software

Moab's reliance on the OSKit created a need for easing the composition of components and regaining any performance losses due to componentization. In pursuit of this goal we designed and developed a new component definition and linking language called Knit, applied it to the OSKit, and explained it in a paper. [12]. Knit helps make C code more understandable and reusable by third parties, helps eliminate much of the performance overhead of componentization, detects subtle errors in component composition that cannot be caught with normal component type systems, and provides a foundation for developing future analyses over C-based components, such as cross-component optimization. The language is especially designed for use with component kits, where standard linking tools provide inadequate support for component configuration. However, Knit is not OSKit-specific, and we have implemented parts of the Click modular router in terms of Knit components to illustrate the expressiveness and flexibility of our language.

4.5 The OSKit

To support this project we made many enhancements to the OSKit [13], including adding network link scheduling, polling network device drivers, real-time thread schedulers, real-time device access, and a port of the OSKit to the StrongARM.

5 Active Network Execution Environments

5.1 ANTS version 2.0

ANTS is a Java-based toolkit for constructing an active network and its applications. We created "ANTSR" (ANTS with Resource control) which became the base for ANTS v2.0. It includes many internal changes for intelligent per-protocol and per-application resource controls. ANTS2 includes a nearly complete restructuring of the ANTS internals, while making few changes to the public ANTS API. It includes a new prototype security subsystem that allows the runtime to make fine-grained access control decisions on a per-application and per-protocol basis. We worked closely with the University of Washington to merge their changes with ours to form ANTS v2.0, which was jointly released. We assumed maintenance of the ANTS software line from then on.

5.2 Bees: A Secure, Resource-Controlled, Java-Based Execution Environment

Mobile code makes it possible for users to define the processing and protocols used to communicate with a remote node, while still allowing the remote administrator to set the terms of interaction with that node. However, mobile code cannot do anything useful without a rich execution environment, and no administrator would install a rich environment that did not also provide strict controls over the resources consumed and accessed by the mobile code.

Based on our experience with ANTS, we developed Bees [14], an execution environment that provides better security, fine-grained control over capsule propagation, simple composition of active protocols, and a more flexible mechanism for interacting with end-user programs. Bees' security comes from a flexible authentication and authorization mechanism, capability-based access to privileged resources, and integration with our custom virtual machine that provides isolation, termination, and resource control. The enhancements to the mobile code environment make it possible to compose a protocol with a number of "helper" protocols. In addition, mobile code can now interact naturally with end-user programs, making it possible to communicate with legacy applications. We believe that these features offer significant improvements over the ANTS execution environment and create a more viable platform for active applications.

6 Active Applications

6.1 Active Protocols for Agile Censor-Resistant Networks

As a demonstration of the benefits of active networking we designed and implemented a prototype application for combating censorship. In a paper [15] and a thesis [16] we argued that content distribution in the face of censorship is a compelling and feasible application of active networking. In the face of a determined and powerful adversary, every fixed protocol can become known and subsequently monitored, blocked, or its member nodes identified and attacked. Frequent and diverse protocol change is key to allowing information to continue to flow. Typically, decentralized and locally-customized protocol evolution is also an important aspect in providing censor-resistance.

A programmable overlay network can provide this type of manually-initiated protocol diversification. We implemented our prototype as an extension to Freenet, a peer-to-peer storage and retrieval system whose goals include censor resistance and anonymity for information publishers and consumers.

6.2 TCP Meets Mobile Code

In joint work with the University of Washington, we also applied active networking principles to transport protocols on end-hosts. In this work we used the "Cyclone" C-like typesafe language and Click router, which we modified to have bounded resource use. In the paper [17] about the ideas and the prototype, we argued that transport protocols such as TCP provide a rare domain in which protocol extensibility by untrusted parties is both valuable and practical. TCP continues to be refined despite more than two decades of progress, and the difficulties due to deployment delays and backwards-compatibility are well-known. Remote extensibility, by which a host can ship the transport protocol code and dynamically load it on another node in the network on a per-connection basis, directly tackles both of these problems. At the same time, the unicast transport protocol domain is much narrower than other domains that use mobile code, such as active networking, which helps to make extensibility feasible. The transport level provides a well understood notion of global safety–TCP friendliness–while local safety can be guaranteed by isolation of per-protocol state and use of recent safe-language technologies. We support these arguments by outlining the design of XTCP, our extensible TCP framework.

6.3 Other Active Applications

We developed an active version [18] of the peer-to-peer protocol underlying the "Doom" networked multi-player distributed simulation game. This project was an experiment in "activating" an existing peer-to-peer protocol, to determine how in-network processing can improve protocol scalability and bandwidth consumption. Non-game nodes route game packets while applying application-level knowledge to packet duplication, aggregation, and compression. For comparison purposes, this "Active Doom" application was implemented thrice: once using the base JanosVM APIs, a second time using ANTS2, and a third time in C.

Integration demonstrations: Twice we integrated our Janos software with that of 3-5 other contractors in the DARPA in the Active Networks Program, joining in large joint technology demonstrations. Our "Team 3" was successful both times.

7 Language and Compiler Technology

Besides the work described above, our work on Kaffe and the JanosVM has included several infrastructure projects.

7.1 Memory Management: The Need for Predictable Garbage Collection

Modern programming languages such as Java are increasingly being used to write systems programs. By "systems programs," we mean programs that provide critical services (compilers), are long-running (Web servers), or have time-critical aspects (databases or query engines). One of the requirements of such programs is predictable behavior. Unfortunately, predictability is often compromised by the presence of garbage collection. Various researchers have examined the feasibility of replacing garbage collection with forms of stack allocation that are more predictable than GC, but the applicability of such research to systems programs had not previously been studied or measured. A particularly promising approach allocates objects in the nth stack frame (instead of just the topmost frame): we call this *deep stack allocation*. In a detailed paper [19] we presented dynamic profiling results for several Java programs to show that deep stack allocation should benefit systems programs, and we described the approach that we are developing to perform deep stack allocation in Java.

7.2 Profiling, Debugging, and Improving the Kaffe Base

We implemented a system for combined profiling of C and JIT'ed Java code in Kaffe. This makes it possible to produce timing and call graph data covering both languages in a format that can be understood by the

GNU “gprof” program. In addition, this work was extended to support run-time debugging of programs in the GNU debugger.

We performed major maintenance on the underlying Kaffe system. We contributed many fixes, enhanced compliance with the Java specification, increased robustness, and improved its threading system. Our changes were continually incorporated into the base version of Kaffe.

Finally, great headway was made on the integration of the GCJ ahead-of-time Java compiler and Kaffe.

8 Emulab/Netbed: an Integrated Network Testbed

The Janos project heavily relied on the facilities provided by our 168-node public network testbed, Emulab, and also helped develop it. Three experimental environments traditionally support network and distributed systems research: network emulators, network simulators, and live networks. The continued use of multiple approaches highlights both the value and inadequacy of each. Netbed, a descendant of Emulab, provides an experimentation facility that integrates these approaches, allowing researchers to configure and access networks composed of emulated, simulated, and wide-area nodes and links. Netbed’s primary goals are ease of use, control, and realism, achieved through consistent use of virtualization and abstraction.

By providing operating system-like services, such as resource allocation and scheduling, and by virtualizing heterogeneous resources, Netbed acts as a virtual machine for network experimentation. This paper presents Netbed’s overall design and implementation and demonstrates its ability to improve experimental automation and efficiency. These, in turn, lead to new methods of experimentation, including automated parameter-space studies within emulation and straightforward comparisons of simulated, emulated, and wide-area scenarios.

Since October 2000 this testbed has been available to researchers 24/7, despite being under constant enhancement. In 2002 we published a paper [20] on its architecture and another on our vision for extending similar benefits to the wireless environment. [21].

References

[1] **Janos: A Java-Oriented OS for Active Network Nodes**. Patrick Tullmann, Mike Hibler, and Jay Lepreau. *IEEE Journal on Selected Areas in Communications*, 19(3):501–510, March 2001.

[2] **Microkernels Meet Recursive Virtual Machines**. Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 137–151. USENIX Association, October 1996.

[3] **Nested Java Processes: OS Structure for Mobile Code**. Patrick Tullmann and Jay Lepreau. In *Proc. of the Eighth ACM SIGOPS European Workshop*, pages 111–117, Sintra, Portugal, September 1998.

[4] **The Alta Operating System**. Patrick A. Tullmann. Master’s thesis, University of Utah, December 1999. 104 pages. Also available at http://www.cs.utah.edu/flux/papers/tullmann-thesis-base.html.

[5] **Techniques for the Design of Java Operating Systems**. Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau. In *Proc. of the 2000 USENIX Annual Technical Conf.*, pages 197–210, San Diego, CA, June 2000. USENIX Association.

[6] **Drawing the Red Line in Java**. Godmar V. Back and Wilson C. Hsieh. In *Proc. of the Seventh Workshop on Hot Topics in Operating Systems*, pages 116–121, Rio Rico, AZ, March 1999. IEEE Computer Society.

[7] **Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java**. Godmar Back, Wilson C. Hsieh, and Jay Lepreau. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 333–346, San Diego, CA, October 2000. USENIX Association.

[8] **Isolation, Resource Management and Sharing in the KaffeOS Java Runtime System**. Godmar V. Back. PhD thesis, University of Utah, Salt Lake City, UT, May 2002.

[9] **An OS Interface for Active Routers**. Larry Peterson, Yitzchak Gottlieb, Mike Hibler, Patrick Tullmann, Jay Lepreau, Stephen Schwab, Hrishikesh Dandekar, Andrew Purtell, and John Hartman. *IEEE Journal on Selected Areas in Communications*, 19(3):473–487, March 2001.

[10] **NodeOS Interface Specification**, Active Network NodeOS Working Group. Available as http://-www.cs.princeton.edu/nsg/papers/nodeos.ps, January 2000.

[11] **Hybrid Resource Control of Active Extensions**. Parveen Patel and Jay Lepreau. In *Proc. of the Sixth IEEE Conf. on Open Architectures and Network Programming (OPENARCH 2003)*, pages 23–31, San Francisco, CA, April 2003.

[12] **Knit: Component Composition for Systems Software**. Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 347–360, San Diego, CA, October 2000. USENIX Association. http://-www.cs.utah.edu/flux/papers/knit-osdi00-base.html.

[13] **The Flux OSKit: A Substrate for OS and Language Research**. Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, October 1997. http://www.cs.utah.edu/flux/-papers/oskit-sosp16.ps.gz.

[14] **Bees: A Secure, Resource-Controlled, Java-Based Execution Environment**. Tim Stack, Eric Eide, and Jay Lepreau. In *Proc. of the Sixth IEEE Conf. on Open Architectures and Network Programming (OPENARCH 2003)*, pages 97–106, San Francisco, CA, April 2003.

[15] **Active Protocols for Agile Censor-Resistant Networks**. Robert Ricci and Jay Lepreau. In *Proc. of the Eighth Workshop on Hot Topics in Operating Systems*. IEEE Computer Society, May 2001.

[16] **Agile Protocols: an Application of Active Networking to Censor–Resistant Publishing Networks**. Robert Ricci. Bachelor's honors thesis, University of Utah, August 2001. 39 pages. http://-www.cs.utah.edu/flux/papers/ricci-thesis-base.htm.

[17] **TCP Meets Mobile Code**. Parveen Patel, David Wetherall, Jay Lepreau, and Andrew Whitaker. In *Proc. of the Ninth Workshop on Hot Topics in Operating Systems*, pages 157–162, Lihue, HI, May 2003.

[18] **Active Doom: Applying Active Networks to Traditional Protocols**. Austin Clements, Patrick Tullmann, and Jay Lepreau. In *18th ACM Symposium on Operating Systems Principles*, Banff, CA, October 2001. Work in Progress presentation.

[19] **The Need for Predictable Garbage Collection**. Alastair Reid, John McCorquodale, Jason Baker, Wilson Hsieh, and Joseph Zachary. In *Second ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 56–63, Atlanta, GA, May 1999.

[20] **An Integrated Experimental Environment for Distributed Systems and Networks**. Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002.

[21] **Lowering the Barrier to Wireless and Mobile Experimentation**. Brian White, Jay Lepreau, and Shashi Guruprasad. In *Proc. HotNets-I*, Princeton, NJ, October 2002.